1

Image Processing

Reid B. Porter

The goal of image processing is to robustly extract useful, high-level information from images and video. The type of high-level information that is useful depends on the application. Examples of applications include object detection and tracking for surveillance, defect detection for automated production systems, and scene classification for remote sensing and map annotation. Extracting high-level information is a difficult research problem and many different algorithms have been suggested.

Image and video processing have been significant application drivers for the reconfigurable computing community since its inception in the early 1990's. Prior to modern reconfigurable devices, image and video processing were also significant application drivers for computer architecture research and VLSI design. There are two related reasons why this application domain has received so much attention in the computer architecture community. The first is the poor, often unacceptable, performance observed in general-purpose processor implementation. This can be attributed to:

- Large volumes of data.
- Exceptionally high memory bandwidth requirements.
- And real-time processing constraints.

The second is the increased, often incredible, performance gains observed in custom or application-specific implementation. This can be attributed to:

- Abundant parallelism in both data and algorithms.
- Local and regular data dependencies.
- Simple fixed point arithmetic and logic operations
- Relatively small bit-widths.

Reconfigurable computers have been used most widely, and successfully, for accelerating *low-level* image processing algorithms. These algorithms are typically applied close to the raw sensor data and are characterized by large data volume. Conceptually, low-level image processing is decomposed into a

processing pipeline with raw image data (taken from a sensor) as input and the desired information as output. Figure 1.1 depicts a typical processing pipeline. Each stage of the pipeline can be a multiple-input, multiple-output transformation.

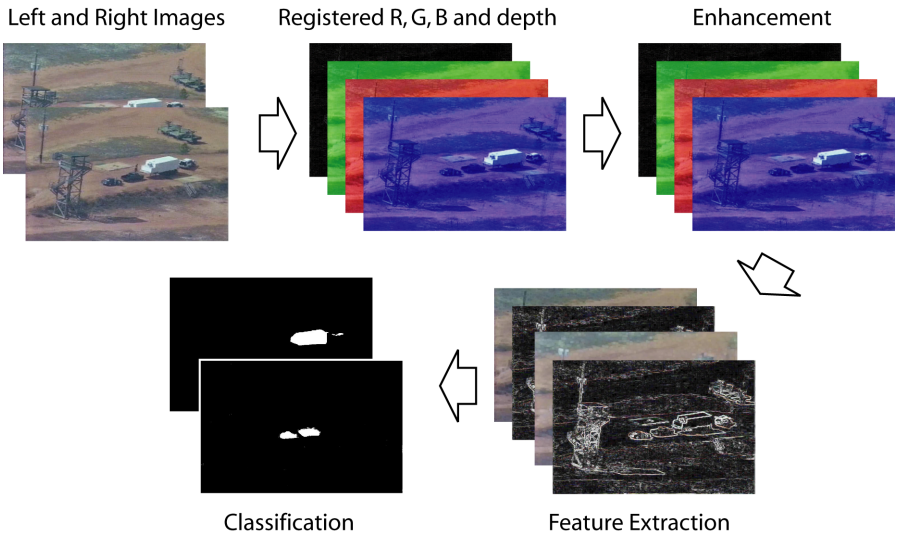

Fig. 1.1. A multi-spectral image processing pipeline

- There may be multiple sources of data from multiple sensors and / or multiple points in time in which case, it can be useful to co-register the data. The relative displacement between data sources is often useful e.g. depth information can be recovered from stereo pairs, and motion information from temporal sequences.
- The image enhancement stage is concerned primarily with removing sensor noise and other environmental variation and attempts to 'enhance' an input image to make subsequent analysis easier.
- Feature extraction is a transformation from the image space, where each pixel usually represents intensity, to a feature space where pixels represent more abstract quantities. These quantities are typically application specific and are chosen to make subsequent processing easier.
- A latter stage of low-level image processing is detection, classification or segmentation, in which an algorithm assigns abstract labels to image pixels. These labels are typically application specific e.g. a non-zero label specifies a region of interest.

Many variants and extensions of this processing pipeline exist for particular applications. In terms of Reconfigurable Computer implementation it is

useful to categorize low-level image processing algorithms based on their data dependencies. Two broad categories of algorithm are:

Local Algorithms: The algorithm depends on data from a relatively small (compared to the image size) neighborhood that is local in spatial and temporal dimensions. Examples include, point or pixel operators (such as Band arithmetic, thresholding), convolution, and motion estimation.

Global Algorithms: The algorithm depends on data from the entire image. Examples include transforms like the fast fourier transform and principle component analysis as well as statistical histogram techniques.

A general rule of thumb for obtaining speed-up with custom computing architectures is to minimize the number of times we access data. By definition global algorithms often require multiple passes through the data and performance compared to general purpose processors is varied and algorithm specific. In this chapter we will concentrate on local algorithms. These algorithms are found in all aspects of the low-level image processing pipeline and they can benefit greatly from Reconfigurable Computer implementation. Reported speed-ups are typically two orders of magnitude compared to general purpose processors.

1.0.1 Local Neighborhood Functions

Local neighborhood functions (also called sliding window functions and spatial filters) are used extensively in image processing and computer vision. These functions are applied at a particular pixel location and their output depends on a finite spatial neighborhood. The function is applied independently at all pixel locations and is typically constant across all pixel locations. Figure 1.2 illustrates the how a neighborhood function is applied for a 3 by 3 neighborhood. When local neighborhood functions are applied at edge locations some of the neighborhood is not defined. The undefined pixels can be assigned a value of 0, or can be assigned the value of the closest pixel. Another common approach is to temporarily increase the size of the input image by reflecting pixel values across each edge.

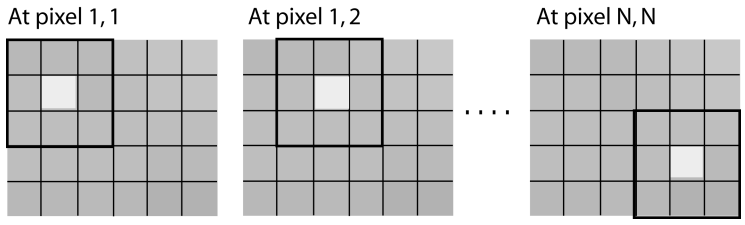

Fig. 1.2. A neighborhood function is applied to all pixels in parallel

The neighborhood function of Figure 1.2 can be generalized in several ways to include a large number of standard image processing algorithms. Neighborhoods can be generated from multiple input images such as color channels or more generally, spectral dimensions. The kernel is 3 dimensional and the neighborhood function slides over the 2 spatial dimensions of the image stack. Point operators such as band arithmetic (average of color channels), clipping, thresholding and pixel scaling can be considered local neighborhood functions when we assume a spatial neighborhood size of 1 pixel. These basic operations are described in detail in most image processing texts [15].

Local neighborhood functions can also receive multiple images in time, and this is typical in video processing applications. This is different to receiving multiple spectral inputs. Similar to FIR (Finite Impulse Response) and IIR (Infinite Impulse Response) filters encountered in signal processing the neighborhood window has a finite temporal extent and slides through time as the function is applied at each time step. The kernel is 3-dimensional and the neighborhood function slides over 3 dimensions (2 spatial and 1 temporal). Note that in the spatial dimension the neighborhood function is applied independently at every location, but for the temporal dimension this is not always the case e.g. neighborhood functions with temporal feedback (IIR).

Local neighborhood functions demand exceptionally high bandwidth to image data. For example, for a modest 3 band 256 pixel wide by 256 pixel high color video sequence, a (typical) 7 by 7 spatial neighborhood size and a 3 frame temporal window, the most general neighborhood function would require access to 441 pixel values at each image location. To obtain real time processing rates at 30 frames per second would require access to approximately 870 million pixels per second. As we will see, most image processing applications are composed of large numbers of local neighborhood functions and therefore the bandwidth requirement quickly exceeds what general purpose computing can provide. Fortunately, due to the regular nature of the memory access across the image array, there are also many opportunities to optimize the memory access. Reconfigurable computers are ideal platforms to tailor memory hierarchies and implement algorithm specific address generation, and therefore great performance gains are possible.

There are two main ways of achieving speedup in local neighborhood functions using Reconfigurable Computers: pixel parallelism and instruction-level parallelism. The two extremes of this approach are illustrated in Figure 1.3.

Cellular Arrays for Pixel Parallelism

Cellular arrays are a natural model for image processing [23]. They consist of an array of cells in two, three or more dimensions. Each cell is associated with an image pixel and each cell has dedicated connections to its local neighborhood. This high-bandwidth local communication is ideal for implementing neighborhood functions; all pixels are processed in parallel, and the entire image is updated in 1 instruction cycle. FPGAs can implement a programmable,

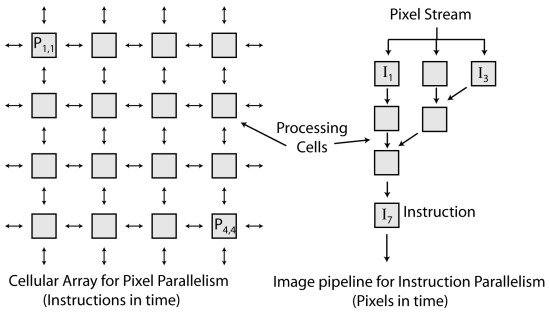

Fig. 1.3. Pixel Parallel verse Instruction Parallel

maximally parallel implementation of a cellular array, but can only efficiently implement a small numbers of cells. Large arrays require multiple FPGAs and/or time multiplexing, and the I/O required to initialize the array read results can dominate the computation time.

Image Pipelines for Instruction-Level Parallelism

In this case only one cell, of the equivalent cellular architecture, is implemented. Data is provided to the cell through a continuous stream of pixels. They are often supplied to the cell one sample at a time, and usually in raster scan order. This arrangement is often suitable for real-time systems where data arrives directly from a serial I/O sensor. Since pixels are processed sequentially, the main way to achieve speed-up for an image pipeline is to execute multiple instructions in parallel. As shown in Figure 1.3 instructions can be implemented either in parallel (increasing the pipeline width) or in series (increasing the pipeline depth). Unlike cellular architectures, accessing a local neighborhood within an image pipeline must be carefully considered. All instructions in a pipeline are being executed at the same time, and therefore it may be difficult to provide data to all instructions at the right time.

We have described the two extremes of a pixel parallel verse instruction parallel design space. In practice any combination of these two extremes may be used. The optimal design point is dictated largely by the memory architecture of the particular FPGA and Reconfigurable Computer. Formalizing and providing tools that can optimize this design choice is a topic of ongoing research [17].

1.0.2 Convolution

Perhaps the most well known local neighborhood function is convolution, which is defined in Equation 1.1. A multiplicative weight W is associated

with each location of the neighborhood $\{k, l\}$, collectively known as the kernel K. The output F of the function is the accumulated weighted sum of the kernel applied at each pixel location $\{i, j\}$.

$$F(i,j) = \sum_{k,l \in K} W_{k,l} * Image(i-k, j-l) \tag{1.1}$$

By selecting the appropriate weights, convolution can implement low-pass, high-pass and band-pass frequency domain filters used extensively in image enhancement and feature extraction. Low-pass filters use positive weights and are used for image smoothing. High pass filters use a kernel with a positive center weight and negative outer weights and are used to enhance high frequency components in an image such as edges and fine detail.

One of the first Reconfigurable Computer implementations of convolution was on the Spash-2 [21]. They used the image pipeline approach and a linear systolic array implementation. Local memory was used to replace multipliers and the lack of on-chip memory meant the image width was limited to 32 pixels. Despite these limited resources the timing for a two 3by3 convolutions applied to a 512 by 512 image was 100 frames per second. A 3 by 3 convolution implementation which is very similar to the linear systolic array is shown in Figure 1.4. The image data is assumed to arrive one pixel each clock cycle in raster scan order. After a fixed latency, this architecture provides access to the entire neighborhood of data every clock cycle.

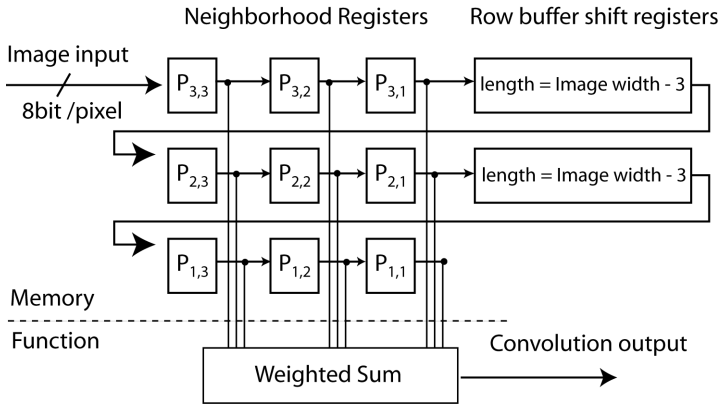

Fig. 1.4. A systolic array for 2-D convolution

The architecture in Figure 1.4 places the lowest demand on external memory bandwidth, but the highest demand on internal memory bandwidth. Each pixel in external memory is accessed only once but for an image width, W, and kernel width, M, the FPGA must store ((M-1)*W + M) pixels on-chip.

Since around 1998, many modern FPGA devices have a the large amount of on-chip memory and this approach has been widely adopted [16], [18], [3].

The length of the shift register in Figure 1.4 depends on the input image. If the image is thousands of pixels wide it is unwise to buffer the entire row. The most common approach is to choose a row length appropriate to the hardware resources at hand (e.g. 64, 128 or 256 pixels), slice the input image into strips of this width, and provide these strips as one long, narrow image to the hardware. Due to the neighborhood, these strips must overlap by a particular number of pixels in order to produce results that *stitch* correctly. This overlap leads to a slight decrease in performance compared to the full length row buffers. Bosi, Bosi and Savaria estimate that dividing a 1024 by 1024 image into 16 slices reduces the number of registers by a factor of 14.8 for a 3 by 3 convolution, while performance is reduced by 6% [4].

When on-chip memory is not available, row-length shift registers may not be possible at all. To maintain the pipeline throughput at one convolution per cycle the design needs to access more than one pixel per clock cycle from external memory. If we can access M neighborhood pixels per cycle we can do without the row length shift registers entirely. For example, if the data-width of external memory is 32 bits and the pixel data width is 8 bits, we can access up to 4 pixels per clock cycle. For a 3 by 3 convolution we need to access three pixels from three different scan lines each clock cycle. Since an image is typically stored in raster scan order in the external memory the memory access must cycle between the three different scan-lines. On-chip registers can be used to buffer the three consecutive pixels from each row and maintain throughput at one convolution per cycle [4].

In the situation just described each pixel is read from memory three times (once for each row in the neighborhood). To reduce the redundant I/O it is possible to implement multiple neighborhood functions, each associated with consecutive rows of the image. The multiple functions exploit pixel parallelism, and also, share local neighborhood access and therefore the I/O is reduced. This approach is described as partial loop unrolling by Draper et. al. with respect to the Single Assignment C compiler [8].

1.0.3 Morphology

The pipelined neighborhood cache in Figure 1.4 can be used for a much wider class of algorithm than just convolution. Mathematical Morphology defines a large family of image processing algorithms, which essentially replace the Weighted Sum function block with a neighborhood order statistic. The kernel for morphological spatial filters is also called a structuring element or region of support and defines the set of pixels from which an order statistic is derived. The shape of the structuring element is very important. The simplest filters are erosion and dilation. Erosion is defined as the minimum from the set of pixels defined by the structuring element and dilation is the maximum. Another popular morphological filter is the median.

Morphological functions are generally far cheaper to implement with digital logic than convolution type functions. First, morphology avoids the multipliers that can become expensive for large neighborhood convolutions. Second, order statistics such as maximum, minimum and median are closely related to the digital domain. The relationship is described by a technique known as threshold decomposition, which was first introduced to analyze the median filter [12]. Threshold decomposition allows gray-valued pixel images to be processed with bit-level hardware and Figure 1.5 illustrates the technique for a 1 dimensional median filter. Pixel inputs are first thresholded at all possible quantization levels; producing a binary *stack* for each input whose height is equivalent to the pixel value. Each quantization level is then processed independently with a positive boolean function. Positive Boolean Functions (PBFs) are a subset of Boolean logic functions in which no input may be negated. To regain a gray-valued output we simply sum the binary outputs from each level. There is a one-to-one correspondence between a PBF and an order statistic, where each logical AND is replaced by a minimum and each logical OR is replaced by a maximum.

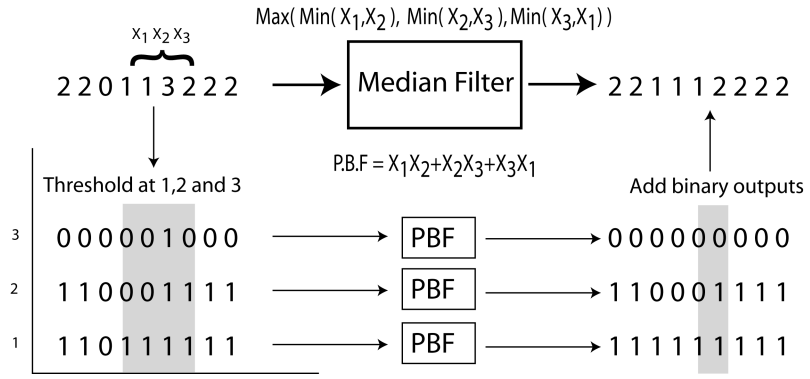

Fig. 1.5. Threshold decomposition for the median function

Threshold decomposition at first appears to have complexity proportional to the number of quantization levels (which may be very high) but this in fact can be reduced to the number of inputs in the filter window. Also, each stack (associated with both inputs and output) can only make a single transition from one to zero i.e. within the same stack ones cannot appear above a zero. This property, known as the stacking property, allows for extremely efficient implementations. Chen proposes a most significant bit first, bit-serial implementation which uses a single PBF [5]. When implementing a 3 by, 3 8-bit pixel, erosion on a Xilinx 6200 series FPGA, Woolfries found that implementing 8 copies of the Chen's implementation used 75% fewer resources and

was 33% faster than implementing the threshold decomposition in Figure 1.5 directly.

The threshold decomposition approach is particular useful in Reconfigurable Computing for implementing order statistics with high complexity, such as the median, and a large number of inputs. For simple order statistics, such as maximum and minimum the number of comparisons is linear in the number of inputs, and a direct sorting network can be implemented efficiently. The direct sort can also be used for the median function if the number of inputs is small [26].

1.0.4 Feature Extraction

We have described the basic neighborhood function building blocks used in image processing. By combining these building blocks in various ways we can implement a large number more complex image processing algorithms that perform Feature Extraction. Feature extraction often has one of two aims:

1. To produce a representation that is invariant to specific image properties such as rotation, illumination, scale etc.
2. To produce a representation suitable for subsequent processing. These quantities often represent things like texture or color, but they can vary greatly depending on the application.

One of the most well known example of feature extraction is edge detection. Asymmetric weight kernels suggested by Roberts, Sobel, Prewitt and Laws estimate image gradients in specific directions. A number of these kernels are used in convolution and the outputs are combined to produce a rotationally invariant edge detection. Outputs are typically combined by the sum of squares, however a sum of absolute values or maximum may be more appropriate in Reconfigurable Computer implementations. Obtaining rotation invariance through multiple kernels is also used in morphology. Figure 1.6a shows an example where a linear structuring is used to *probe* the image for linear image features such as roads. A maximum is used to combine multiple outputs during dilation, and a minimum is used during erosion. For the image processing pipeline architecture, multiple rotations correspond to increased pipeline width. Considerable memory resources can be saved if multiple rotations share row buffers and neighborhood registers.

Gabor filters provide are another example of where we need to increase the image pipeline width. The Gabor kernel is defined as a complex plane wave modulated by a Gaussian distribution. It implements a band-pass frequency domain filter. For feature extraction, a bank of Gabor filters are implemented, each tuned to specific spatial frequencies and orientations. The quantity used in subsequent processing is often the magnitude of the complex convolution which exhibits invariance to small shifts of the input image. The number of filters in a Gabor filter bank can be quite large. In this case, it may be more

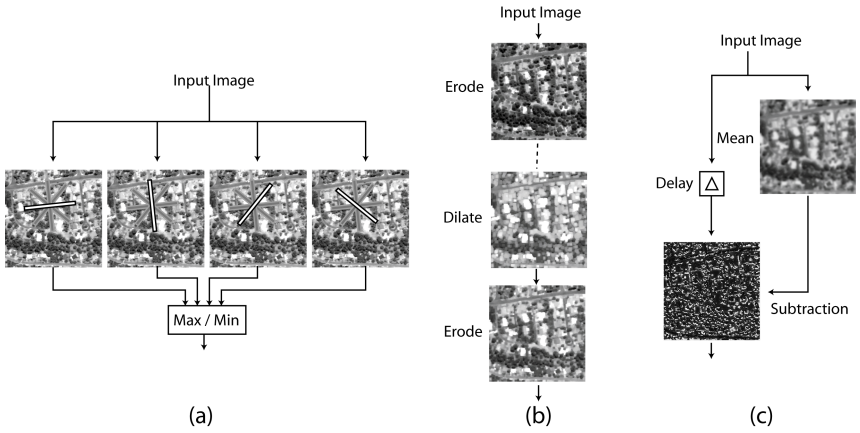

Fig. 1.6. Examples of feature extraction: a) rotationally invariant linear features b) deep morphological pipelines and c) adaptive thresholding

efficient to implement convolutions in the frequency domain. This requires a Fast Fourier Transform (FFT) and an Inverse FFT, which are available for most modern FPGAs as third part cores[27].

Many complex morphological algorithms for feature extraction such as opening, closing, open-close and close-open filters are built by successive application of erosion and dilation. Usually the shape of the structuring element is constant between successive erosions and dilations. As shown in Figure 1.6b these algorithms can be implemented in the image pipeline architecture by simply increasing the pipeline depth.

Another class of feature extraction algorithms are locally adaptive, which means that a neighborhood function is dependent upon some statistic of the local neighborhood. A popular example is adaptive thresholding in which the centre pixel is thresholded by the mean or median value of the neighborhood. Locally adaptive functions define small sub-trees within an image processing pipeline. For implementation within the image pipeline different pipeline paths must be latency adjusted before they can be combined pixel-wise. The adaptive threshold example is illustrated in Figure 1.6c.

There are many other examples in feature extraction where algorithms are implemented by cascading multiple local neighborhood functions. Reconfigurable computers gain a significant advantage over general purpose machines for these types of algorithms. Apart from increased latency (which in many applications is not important), the pipeline throughput is constant at one pixel per cycle. The FPGA resources limit how far this approach can be taken. Depending on the FPGA architecture and specific type of algorithm, this can be logic limited or memory limited. Once this limit is reached, multiple passes of the data will be required to execute further instructions.

1.0.5 Automatic Target Recognition

One application where FPGA resources are often not sufficient to implement the maximal throughput pipeline is template matching for Automatic Target Recognition. In template matching the neighborhood function kernel is a small sub-image from the original image or from a related image and the neighborhood function calculates a distance metric, like correlation, between the sub-image and the original image at all pixel locations. Typically there are a large number of templates, or kernels, and the problem is to find the template with the best match at each pixel location. In practical systems the number of templates often far exceeds what can be matched in parallel with Reconfigurable Hardware. For example, an ATR algorithm developed for synthetic aperture radar by Sandia National Laboratory has approximately 5700 templates associated with each target. With tens or hundreds of targets, it becomes clear that a practical implementation will require a number of passes.

The most efficient hardware utilization is gained by customizing the FPGA for each pass with the configuration bit-stream. This is often appropriate when each pass of the data performs a significantly different type of processing, however the approach can also be used to generate optimal specializations of a generic pipeline. For example, Chia et.al. have produced an ATR system called Mojave that produces specialized matching circuits for different templates [6]. They call their approach partial evaluation, and it exploits several properties of the Sandia application:

- The templates are sparse so not all neighborhood pixels are involved in the correlation. Chia et. al. estimate that for approximately half the templates this approach uses 5.8% of the resources used in a general purpose circuit.
- Many templates share common pixels and therefore share partial results in the correlation.

The Mojave system provides a number of CAD tools that can automatically perform the above optimizations for a given set of templates. The system matches 8 by 8 templates against a 128 by 128 video image and was able to achieve an improvement factor between 2 and 10 over the existing ASIC implementation. Device reconfiguration is an attractive approach to multiple pass image processing. The approach is unique to Reconfigurable Computing and it can lead to significant performance improvements. One disadvantage of the approach is that it depends on being able to rapidly reconfigure the FPGA. The Mojave system is based on the Xilinx 4013PG233-4 FPGA which requires 30ms to reconfigure. In comparison, the FPGA processes 4 templates in parallel and takes 16ms for 1 pass. The net result is a system that takes 46ms to evaluate 4 templates.

Another way to implement multiple pass hardware specializations is with partial reconfiguration. In many image processing applications, the image pipeline can have very similar implementation requirements from one pass

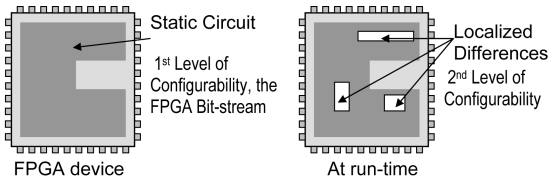

Fig. 1.7. Localized reconfiguration.

to the next. Computations are often typically regular this means implementation difference can be localized and reconfiguration time minimized. Figure 1.7 illustrates the concept. This approach was used for the Sandia ATR application by Bellows and Hutchings [2]. They targeted the rapidly reconfigurable Xilinx XC6200 series FPGA. Using placement constraints they arranged a 2-D systolic array of processors with static interconnect. At run-time the function of these processors is specialized based on a particular template that is being matched. The hardware efficiency of this approach can be very close to that achieved by a complete re-synthesis. The disadvantage of the approach is tied to the limited partial reconfiguration capabilities of most commercial FPGAs. For example, to manipulate the routing to select which inputs are supplied to a neighborhood function is difficult with most FPGA devices.

The alternative to FPGA based partial reconfiguration is to build the multi-pass variability into the hardware design itself. This involves increasing the complexity of the design to include the required variability, and provide on-chip configuration registers with an appropriate interface. Rencher and Hutchings used this approach when implementing the Sandia ATR application on the Splash 2. They implemented a single general purpose matching algorithm for a 16 by 16 template. Each template was loaded from local memory to on-chip registers where it was matched with the input image using a deep image processing pipeline. A control circuit monitored the match from each pass and maintained a record of the best template. Rencher and Hutchings estimated their design running at 13.2 MHz outperformed a HP 770 running at 110 MHz by two orders of magnitude. Building custom configuration circuits can be extended to any level of flexibility e.g. configuration registers can store program instructions for an arithmetic logic unit or a micro-controller. Typically, this approach will be most resource efficient when the multi-pass variability is localized and configuration circuits are tailored to the problem at hand.

1.0.6 Image Matching

Area based image matching is another class of local neighborhood algorithm that is used extensively in low level image processing. With stereo cameras, two cameras are used to image a scene from two different locations so that

a physical point appears in different locations in each camera image. From the difference in location (called the disparity), the depth of the point can be calculated. In video cameras, object or camera motion produces a similar effect and the difference in location (called the displacement) can be used to estimate a motion vector. The image matching problem is find the corresponding points in each image. In area based matching techniques, a point to be matched becomes the centre of a neighborhood. The matching problem involves finding a similarly sized neighborhood in the second image that is the best match for the neighborhood in the first image. Figure 1.8a illustrates the matching problem for a single pixel. The procedure is repeated for every pixel in the template image.

Some popular metrics for matching include the sum of squared (SSD) and sum of absolute differences (SAD), as well as the normalized cross correlation (NCC):

$$F(i,j) = \frac{\sum_{(k,l)\in K} W_{k,l} * Image(i-k, j-l)}{\sqrt{\sum_{(k,l)\in K} W_{k,l}^2 * \sum_{(k,l)\in K} Image^2(i-k, j-l)}} \tag{1.2}$$

Several metrics have been suggested that aim to provide the accuracy of NCC with less expense. A method that is particularly appropriate in FPGA implementations is to use the relative ordering of the pixel intensities to calculate similarity [30]. Images are first transformed according to local neighborhoods. In the *rank transform* each pixel intensity is replaced by an integer that represents the number of pixels within a neighborhood whose value is less than the centre pixel. The *census* transform replaces each pixel with a bit string which encodes the neighborhood pixels according to their location. If a pixel value is less than the centre pixel the corresponding position in the bit string is set to 1, otherwise it is set to 0. Once the images have been transformed, points are matched by using the traditional area based methods. The rank transform typically uses the SAD or SSD similarity metric while the census uses a metric based on the hamming distance between the two bit vectors. We estimate the rank matching metric consumes approximately 50% fewer resources than SSD and at least 75% fewer resources than NCC. This is mainly due to the smaller data width of the rank metric output[19].

Image matching and template matching are in some ways similar. In both algorithms there are a very large number of templates, and the problem is to find the template with the best match. There are also two significant differences:

1. For image matching the template has a search window that is typically much smaller than the original image. In template matching each template is matched at every location in the entire image.
2. In image matching the templates are local neighborhoods taken from every pixel location in the template image, which means consecutive templates have overlapping values. In template matching each template may

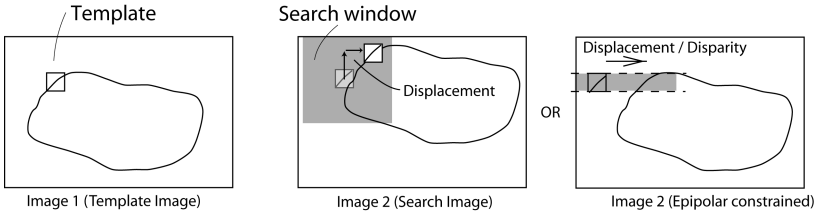

Fig. 1.8. Searching for the best match in a) the general case and b) in the epipolar constrained case

be completely different from every other template. When templates do have overlap (as in the Sandia application) it is template specific.

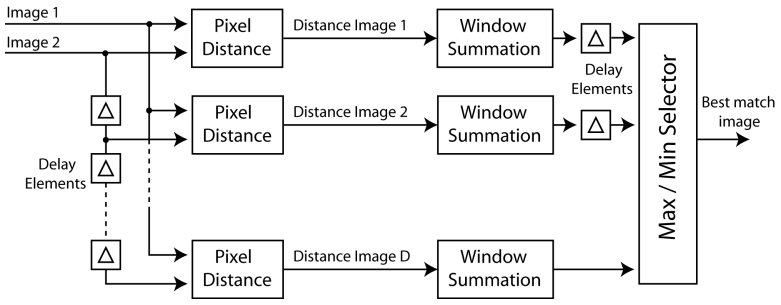

Fig. 1.9. A real-time matching architecture.

These differences suggest an alternative approach to implementation for image matching which is illustrated in Figure 1.9. The two image pixels streams arrive at the same time but then are displaced from one another by varying degrees via delay elements. Images can be displaced horizontally via registers, but require row-length shift registers to be displaced vertically. Often in stereo matching the two cameras are mounted carefully to ensure that the two scan lines are in correspondence. If this is the case then the displacement can be assumed to lie on the same horizontal line which greatly reduces the search window as illustrated in Figure 1.8b.

Most metrics used in matching, such as SAD, SSD, NCC and the Hamming distance, are based on a neighborhood summation which can be calculated in two steps. We first we calculate a distance image based on a pixel wise distance metric between the two displaced images. We then accumulate the distance image within a local neighborhood. This can be computed with the convolution function as in Figure 1.4, or since there are redundant additions (due to equal weights in the convolution), it can be computed with running

totals [11]. Neighborhood summation with running totals allows much larger neighborhoods to be accumulated and is a two-stage process:

1. Calculate row sums: A new row sum is calculated from the previous row sum by adding the new pixel and subtracting the last pixel. The row sum calculation is easily pipelined with a neighborhood row shift register and an adder / subtractor.
2. Calculate column sums: This is similar to the first step but instead of accumulating and subtracting pixels we accumulate and subtract row sums. The number of running totals is equal to the image width. Since the row sums are being calculated in scan line order a large shift register is required to subtract the last row sum within the pipeline. One way to avoid this shift register is to introduce redundant additions so that the last row sum is calculated at the same time as the first row sum [10]. Figure 1.10 illustrates the main components required in calculating the column sums. The running totals are kept in memory and accessed sequentially as new row sums are generated.

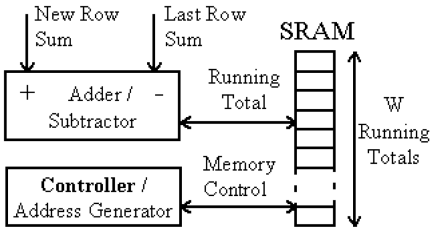

Fig. 1.10. Calculating the running totals.

To produce the final output the neighborhood sums from the various distance images are compared. The displacement with the smallest sum (except for the NCC metric for which we choose the largest) corresponds to the best match in the search area. The implementation described has been used by several researchers to obtain real-time depth maps from stereo cameras [10], [28]. As far as we know the architecture has not been used for motion estimation on FPGAs. This is probably because of the large amount of memory that is required to search for vertically displaced neighborhoods, which until recently would have made real-time implementation infeasible. In addition, most Reconfigurable Computer implementations for motion estimation target block matching algorithms used in video compression [22]. In these algorithms matching neighborhoods are usually non-overlapping which means only a subset of the pixels within the template image are matched. This leads to different opportunities for optimization and therefore different implementations.

1.0.7 Evolutionary Image Processing

In image processing we often define an error, or loss function that measures how well a particular algorithm solves the problem of interest. The task is then to find, through optimization, the algorithm that minimizes this loss function and is therefore in some sense *optimal*. Optimal image processing algorithms are generally able to outperform fixed algorithms since they are tuned for the specific data and task at hand. Many of the standard image processing algorithms, such as convolution, morphology and matching are used in optimal image processing e.g. in optimal image enhancement we replace a fixed convolution kernel like Gaussian smoothing with convolution weights that are optimized to minimize a mean squared error. Another much studied optimal image processing problem is pattern recognition where the error function is based on detection and false alarm rates.

Reconfigurable Computers are particularly useful in implementing optimization problems since the implementation requirements of optimization problems vary greatly from one problem to the next [1]. Evolutionary Algorithms (EA), define a family of optimization techniques for which this is particularly true. EA include genetic algorithms, genetic programming, evolutionary programming and evolutionary strategies. They are one of the most flexible optimization techniques in use today, and have been applied to a variety of research, industrial and commercial problem solving activities [7]. EA optimization is based on sample and test. A large number of candidate solutions are generated randomly. Each candidate is evaluated and assigned *fitness* by applying the solution to a training set and calculating the loss function. Based on this fitness, the population of candidate solutions is resampled, and the process repeats until candidates achieve a desired level of performance. EA can be applied to many problems in image processing but it is very computationally intensive. Each candidate evaluation is typically a complete pass of an image processing pipeline, and a large number of evaluations are required.

One of the most effective ways to use a Reconfigurable Computer for evolutionary image processing is as a fitness evaluator. The basic architecture is shown in Figure 1.11. An application specific image pipeline is implemented in much the same way as in conventional image processing. We then add a simple control structure that compares the pipeline output to a desired output and calculates the error function. Note, if the pipeline exploits pixel parallelism it is likely that multiple error functions will be implemented in parallel. The pipeline must be used to evaluate many different candidates and therefore it requires a second level of configurability appropriate to the optimization problem. Similar to the ATR example, this can be implemented with partial reconfiguration, and / or with custom configuration circuits as illustrated in Figure 1.11.

Apart from fitness evaluation, the evolutionary algorithm itself is a very simple algorithm and could be implemented onboard the Reconfigurable Computer. Sidhu et.al. describe a genetic programming pipeline implemented on a

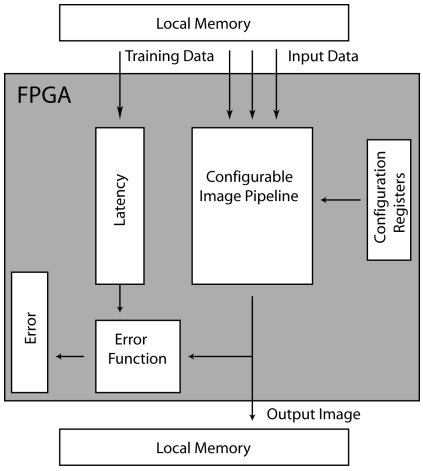

Fig. 1.11. A fitness evaluator architecture for evolutionary image processing.

XC6264 FPGA which obtains speed-up of 19 compared to a 200MHz Pentium Pro for an arithmetic regression problem, and three orders of magnitude for a logic-based multiplexer problem [25]. For the image processing application domain the computation time for the fitness evaluation usually far exceeds the computation time of the EA. In a co-processor environment, where the Reconfigurable Computer sits on a host processor bus, it is typically not necessary to implement the EA in hardware.

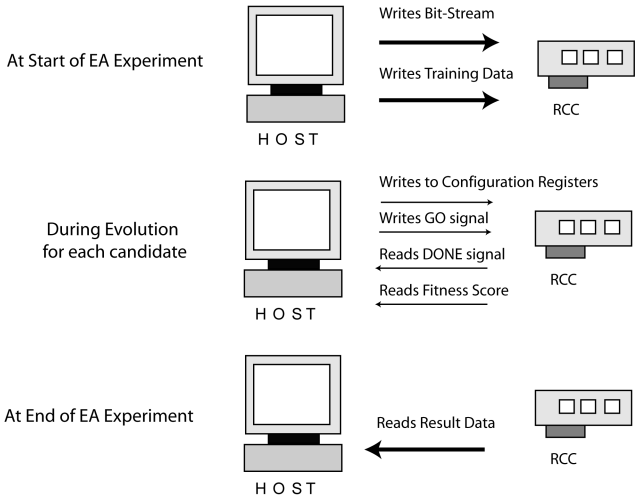

Fig. 1.12. Hardware / software portioning for evolutionary image processing.

Using the Reconfigurable Computer as a co-processor generally implies a low bandwidth connection between the reconfigurable computer and the host processor. To minimize communication across this connection it is important to calculate the error function on chip. Figure 1.12 illustrates the ideal arrangement. Large volume input data and training data are loaded once at the start of optimization to the RCC local memory. Communication between host and the RCC during optimization involves writing to on-chip pipeline configuration registers, initiating the pipeline evaluation, and then retrieving the output error. Only at the end of optimization, is the result image from the lowest error pipeline retrieved for inspection.

Recently, in a field known as evolvable hardware, we observe an interesting consequence of approaching image processing as an optimization problem. The idea is to use evolutionary algorithms to explore non-tradition parameterizations of image processing problems to produce solutions with more efficient FPGA implementations. One of the first applications of evolvable hardware to image processing involved optimizing a variable-length encoded PLD AND-OR array to solve a binary character recognition problem[14]. Many other examples of this approach have now been published [9], [29], [24]. Evolvable hardware researchers have devloped many novel image processing algorithms by optimizing collections of low-level building blocks similar to FPGA logic cells. This approach can produce extremely compact solutions, but will produce little speed-up over general purpose machines unless a large number of these functions are implemented in parallel. This observation led us to develop a system called Pooka, which combines evolutionary image processing with a Reconfigurable Computer coprocessor to solve scene classification and terrain mapping problems in satellite imagery [20]. In the Pooka system we explore a much more abstract parameterization of neighborhood functions, and focus on finding solutions that combine multiple copies of these functions within a deep processing pipeline. The net result is a system that can solve complex practical problems and obtain significant speed-up compared to a general purpose processor.

The Pooka pipeline is illustrated in Figure 1.13. There are 18 highly pipelined functions (or layers): 9 of these functions are used to combine multiple spectral channels and their spatial neighborhood is one pixel. The remaining 9 functions implement functions of a 5 by 5 neighborhood. The pipeline can have up to 16 different inputs. In Figure 1.13 the input imagery has 4 spectral channels but in multi-spectral imagery this can be much larger. The connectivity at the pipeline input and between processing layers is made configurable through large multiplexers which are controlled by the on-chip configuration registers. The basic building block within the Pooka system has two inputs (a and b) and one output which are all 8 bit 2s complement integers. A three bit configuration register dictates which one of eight functions the building block implements. These functions are summarized in Table 1.1.

In each spatial layer there are 24 configurable building blocks. In each spectral layer there are 3 configurable building blocks. The connectivity be-

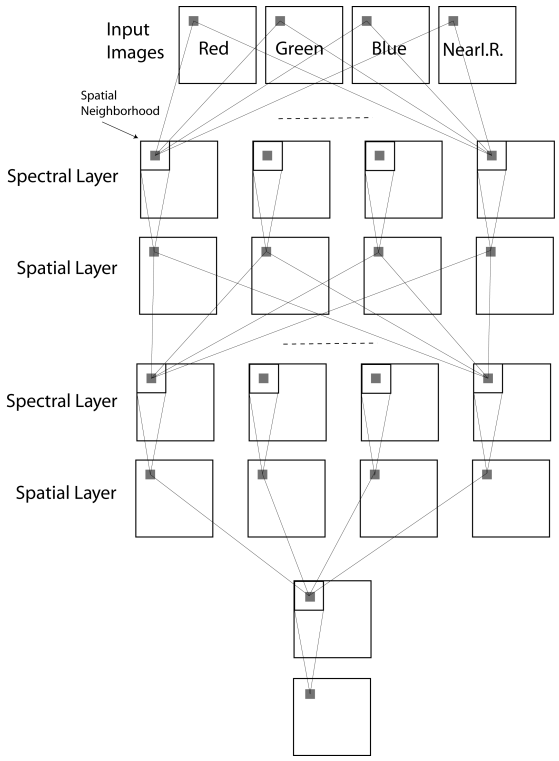

Fig. 1.13. A 18 layer image pipeline for multi-spectral image classification.

Function	Operation
Average	$\frac{a+b}{2}$
Difference	$\frac{a-b}{2}$
Absolute Average	$\lvert\frac{a+b}{2}\rvert$
Absolute Difference	$\lvert\frac{a-b}{2}\rvert$
Maximum	$Max\{a,b\}$
Minimum	$Min\{a,b\}$
Select Left	a
Select Right	b

Table 1.1. The Pooka configurable building block.

tween blocks is largely hard coded and is described with other implementation details in Porter et. al . [20]. We implemented the Pooka system on a Firebird reconfigurable computer from Annapolis Microsystems [13]. This is a 64-bit, 66MHz PCI co-processor that contains a Xilinx Virtex 2000E FPGA, and a total of 36Mbytes of on-board memory distributed in 5 independent banks. The 18-layer network used 64% of the FPGA logic and 35% of the block ram post place and route and could be clocked at 50MHz.

During evolution the Pooka system obtains speed-up of three orders of magnitude over a software simulation running on a 500MHz Pentium III workstation. This is attributed to the fact that Pooka is based on a configurable building block that is efficiently implemented in hardware but inefficiently in software. In hardware, the configurable building block network is completely pipelined and is carefully hand designed to make the best use of Virtex FPGA resources. In software, the 24 configurable building block network requires many conditional assignments, all within nested loops. The software compiler has few optimizations available to it and the performance is poor. A possibly more meaningful measure of performance can be estimated by considering a high-level approximation of Pooka components. For each Spectral layer in the pipeline, a linear combination is calculated. For each Spatial layer, a 5 by 5 neighborhood average is calculated. The relative speed-up in this case was estimated at two orders of magnitude, however the software implementation is slightly simpler (but has greater bit-widths) than the Pooka implementation.

The speed-up achieved by Reconfigurable co-processors in evolutionary image processing can be close to that achieved in real-time reconfigurable computing systems. This is because there is very little communication between the host and Reconfigurable Computer during optimization. Image data is always on-time (since it is stored in local memory) and the image pipeline operates almost continuously at peak capacity. However, once the optimization is complete, the performance of the optimized image pipeline in application will depend on several factors which are typically related to data I/O. In the Pooka system, optimized pipelines are typically applied to large satellite images (2 to 4 GigaByte images) that are stored on the host computer hard disk. Therefore, the execution time for the Pooka system must include the time required to read data and write results to disk, as well as the time to transfer data to and from the coprocessor across the PCI bus. We found this overhead reduced the 100x speed-up by a factor of 10.

1.1 Summary

The local, regular nature of local neighborhood functions, which are used extensively in image processing, provide many opportunities to exploit parallelism. Image data is inherently parallel, and local neighborhoods have considerable overlap from one pixel to the next. Image algorithms are inherently parallel and are often implemented with long sequences of basic operations.

Combined, this means hardware engineers have a rich design space with many degrees of freedom. In many ways, the high performance implementations of image processing algorithms that have been reported are due to this flexibility in design space. The hardware engineer can choose the type and level or parallelization appropriate to the Reconfigurable Computer at hand based on number of gates and on-chip / off-chip memory bandwidth. Not many other applications have this luxury.

In this chapter we have described *prototypical* architectural solutions to several image processing problems. In practice, the implementation details of these implementations for specific Reconfigurable Computers can greatly affect performance, and therefore, exploring the design space with the specific resource constraints is very important. Optimization under resource constraints is often what makes *hard*ware design *hard* for humans. Given the many degrees of freedom in image processing, optimal solutions are likely to remain *hard* for automated resource allocation tools and techniques as well. As computational capacity and memory bandwidth increase, we speculate that non-optimal, but sufficient solutions will become acceptable, simplifying the problem for both humans and machine.

References

1. D. Abramson, A.d. Silva, M. Randall, and A. Posutla, *Special purpose computer architectures for high speed optimisation*, Second Australasian Conference on Parallel and Real Time Systems, 1995.
2. Peter Bellows and Brad Hutchings, *Designing run-time reconfigurable systems with jhdl*, Journal of VLSI Signal Processing **28** (2001).
3. K. Benkrid, D. Crookes, and A. Benkrid, *Towards a general framework for fpga based image processing using hardware skeletons*, Parallel Computing **28** (2002).
4. Bernard Bosi, Guy Bois, and Yvon Savaria, *Reconfigurable pipelined 2-d convolvers for fast digital signal processing*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems **7** (1999), no. 3.
5. Keping Chen, *Bit-serial realizations of a class of nonlinear filters based on positive boolean functions*, IEEE Transactions on Circuits and Systems **36** (1989), no. 6.
6. Kang-Ngee Chia, Hea Joung Kim, Shane Lansing, William H. Mangione-Smith, and John Villasenor, *High-performance automatic target recognition through data-specific vlsi*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems **6** (1998), no. 3.
7. D. Dasgupta and Z. Michalewicz, *Evolutionary algorithms in engineering applications*, Springer-Verlag, Berlin, 1997.
8. Bruce A. Draper, Ross Beveridge, A.P. Willem Böhm, Charles Ross, and Monica Chawathe, *Accelerated image processing on fpgas*, IEEE Transactions on Image Processing **12** (2003), no. 12.
9. J. Dumoulin, J.A. Foster, J.F. Frenzel, and S. McGrew, *Special purpose image convolution with evolvable hardware*, Workshop on Real-World Applications of Evolutionary Computing, Lecture Notes in Computer Science, vol. 1803, 2000.
10. P. Dunn and P. Corke, *Real-time stereopsis using fpgas*, 7th International Workshop on Field-Programmable Logic and Applications: FPL '97, September 1997.
11. Olivier Faugeras, Thierry Vieville, E. Theron, J. Vuillemin, B. Hotz, Z. Zhang, L. Moll, P. Bertin, H. Mathieu, P. Fua amd G. Berry, and C. Proy, *Real-time correlation-based stereo: Algorithm, implementations and applications*, Technical Report 2013, INRIA, 1993.
12. J.P. Fitch, E.J. Coyle, and N.C. Gallagher, *Median filtering by threshold decomposition*, IEEE Trans. Acoust., Speech, Signal Processing **ASSP-32** (1984), no. 6.

13. Annapolis Micro Systems Inc., *Firebird reconfigurable computer*, http://www.annapmicro.com, 2004.
14. M. Iwata, I. Kajitani, H. Yamada, H. Iba, and T. Higuchi, *A pattern recognition system using evolvable hardware*, Proceedings of Parallel Problem Solving from Nature IV - PPSN IV, 1996.
15. A. K. Jain, *Fundamentals of digital image processing*, Pretice Hall Information and System Sciences Series, Pretice Hall, New Jersey, 1989.
16. Hongtu Jiang and V. Owall, *Fpga implementation of real-time image convolutions with three level of memory hierarchy*, IEEE International Conference on Field-Programmable Technology (FPT), December 2003.
17. Xuejun Liang and Jack Shiann-Ning Jean, *Mapping of generalized template matching onto reconfigurable computers*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems **11** (2003), no. 3.
18. Z. Nagy and P. Szolgay, *Configurable multilayer cnn-um emulator on fpga*, IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications **40** (2005), no. 6.
19. R.B. Porter and N.W. Bergmann, *A generic implementation framework for fpga based stereo matching*, Proceedings of IEEE TENCON '97: Speech and Image Technologies for Computing and Telecommunications, December 1997.
20. Reid B. Porter, N. Harvey, S. Perkins, J. Theiler, S. Brumby, J. Block, M. Gokhale, and J. Szymanski, *Optimizing digital hardware perceptrons for multi-spectral image classification*, Journal of Mathematical Imaging and Vision **19** (2003).
21. Nalini K. Ratha, Anil K. Jain, and Diane T. Rover, *Convolution on splash 2*, Proceedings of the 1995 IEEE Symposium on FPGAs for Custom Computing Machines, 1995.
22. N. Roma and L. Sousa, *Automatic synthesis of motion estimation processors based on a new class of hardware architectures*, Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology **34** (2003), no. 3.
23. A. Rosenfeld, *Parallel image processing using cellular arrays*, Computer **16** (1983).
24. L. Sekanina, *Image filter design with evolvable hardware*, EvoWorkshops 2002 Conference, Lecture Notes in Computer Science, vol. 2279, 2002.
25. R.P.S. Sidhu, A. Mei, and V.K. Prasanna, *Genetic programming using self-reconfigurable fpgas*, 9th International Workshop on Field Programmable Logic and Applications, FPL99, 1999.
26. John L. Smith, *Implementing median filters in xc4000e fpgas*, http://www.xilinx.com/xcell/xl23/xl23_16.pdf, 2004.
27. N. Voss and B. Mertsching, *Design and implementation of an accelerated gabor filter bank using parallel hardware*, 11th International Conference on Field Programmable Logic and Applications, August 2001.
28. J. Woodfill and B. Herzen, *Real-time stereo vision on the PARTS reconfigurable computer*, IEEE Workshop in FPGAs for Custom Computing Machines, 1993.
29. M. Yasunaga, T. Nakamura, J.H. Kim, and I. Yoshihara, *Kernel-based pattern recognition hardware: its design methodology using evolved truth tables*, The Second NASA/DoD Workshop on Evolvable Hardware, 2000.
30. R. Zabih and J. Woodfill, *Non-parametric local transforms for computing visual correspondence*, 3rd European Conference on Computer Vision, 1994.